CHAPTER 12
I LIKE THE SOUND OF THAT

Audio on the PC has come a long way from the beeps and chirps of the first IBM PC. Sound was initially designed into the PC platform as an alerting mechanism, so the obnoxious noise was warranted. Today, however, the sound quality of the PC is comparable with high-end consumer audio equipment.

USB was designed to support sound very well. The isochronous transfer type was specifically designed into USB to support audio and other time-dependent data transfers. This is the first time in the book that we are going to use isochronous transfers so we need to slow down a little to learn a little more theory before embarking on our first example. A generic class driver is available, the audio class, and its inclusion in the operating system means that there is no OS code to write. Dealing with time-dependent data in our I/O device however, is not as straightforward as the HID examples. We will start with the simplest audio device, an 8bit mono speaker and progress through various designs including a telephone. The chapter will conclude with some MIDI designs and will show how USB is giving this standard a new applications range.

DESCRIBING HARDWARE USING DESCRIPTORS

There is an enormous variety of electronics circuitry used to create audio equipment and, if we are to use the PC host to contribute to an audio solution, then we must be able to describe this hardware in terms that the PC host can understand. The Audio Class defines two types of class-specific descriptors; Audio Control descriptors are used to construct a software model of the physical audio hardware and Audio Streaming descriptors are used to describe the format of audio data flowing through this model. An Audio Class device will therefore always have two interfaces - an Audio Control interface and an Audio Streaming interface.

Audio Control Interface

The requirement to describe physical entities such as signal inputs, signal outputs, selector switches, mixers, and other controls created a software model that defined these entities as standard **terminals** or **units**. A connected collection

of terminals and units can be used to describe any audio device. A descriptor is defined for each terminal and unit type.

An **Input Terminal** describes the source of an audio input signal; this could be a "real-world" analog signal or a digital audio input stream.

An **Output Terminal** describes the destination of an audio output signal; this could be a "real-world" analog signal or a digital audio output stream.

A **Mixer Unit** transforms a number of input channels into a number of output channels (typically only one output channel), while a **Selector Unit** directs one of several input channels to an output channel.

A **Feature Unit** provides basic control over the incoming logical channel. It provides features such as volume, mute, tone, and graphic equalization. More sophisticated controls are implemented by a **Processing Unit, Dolby Prologic Unit, Stereo Extender Unit, Reverberation Unit,** or other units. Figure 12.1 shows some of the building block icons used to model the physical audio entity.

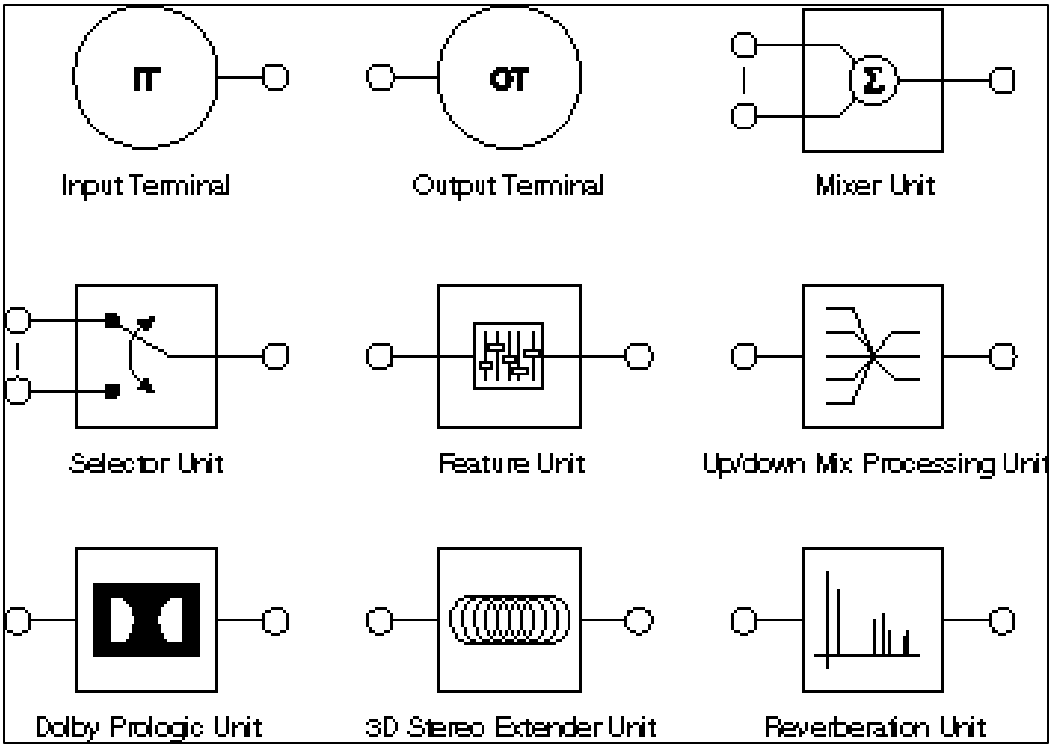


Figure 12-1. Some of the Icons used to model an audio device

This is a little abstract, so let's get into the first example to see how these descriptors are used. We will design a simple USB I/O device that will drive a speaker. For simplicity we shall use single 8-bit signal with a sampling rate of 16KHz. Later examples will add higher resolution, more signals and higher sampling rates but lets use a simple case to learn the theory! This first example uses an EZ-USB microcontroller (we can reuse a lot of the software we created in Chapter 7), an 8-bit Digital-to-Analog converter and a simple audio amplifier driving a small speaker as shown in Figure 12-2. From the I/O device's perspective it will receive 16 bytes every frame which it must output to the DAC at a rate of 1 byte every 64 usec.

Figure 12-2. Schematic and hardware for first example.

Our first task is to create a model for this hardware. This bare-bones example has no features: it simply converts the data sent to it by the PC host into audible sounds. This results in a simple model for the Audio Control as shown in Figure 12-3. Note that each entity is given a unique ID and will have an associated descriptor.

Figure 12-3. Audio Control for speaker example.

The hardware described in Figures 12-2 and 12-3 is now put into a series of Audio Control descriptors as shown in Figures 12-4. Figure 12-4 shows the descriptor hierarchy and the detailed information is added in a later figure.

Figure 12-4. Audio Control Descriptors.

Audio Streaming Interface

We learnt in Chapter 2 that USB reserves a portion of the bus bandwidth for isochronous transactions. An isochronous I/O device also has a responsibility here – when it first enumerates it should request ZERO bandwidth. Since it requests no bandwidth it will successfully enumerate whatever the current load on USB. When an application program needs to use this isochronous I/O device it will request its desired bandwidth requirement. If this bandwidth is available then the I/O device is activated and guaranteed to get the requested bandwidth. If the requested bandwidth is not available then the I/O device is interrogated to discover if it can operate at a lower bandwidth requirement – if it can and there is available bandwidth then it is granted access else it is not activated. The isochronous I/O device provides this information via Alternate Configurations that request less and less of the USB bandwidth. As soon as the isochronous transfer is over the I/O device should release its allocation of bus bandwidth so that other devices have the opportunity to use it. Figure 12-5 shows the hierarchy of Audio Streaming Descriptors.

Figure 12-5. Audio Streaming Descriptors

These two sets of interface descriptors are concatenated with a Device Descriptor, Configuration Descriptor and Strings Descriptor to produce a complete speaker example as shown in Figure 12-6.

Figure 12-6. Descriptors for Speaker Example.

IMPLEMENTING THE I/O DEVICE FIRMWARE

Now that we have built the descriptor tables we can review what else needs to be done in this example. It is, in fact, quite similar to the Buttons and Lights example of Chapter 7 and most of the firmware that we developed will be reusable.

All of the standard request handling can be "cut and pasted" into this example. The CLASS request handling will be different since we are now AUDIO class and not HID class; we have no features in this first example so the PC host will not send any class requests – we can mark them all as INVALID in the CommandTable.

We will receive 16 bytes every frame on endpoint 1 and these must be metered out to the DAC at 64 usec intervals. This is similar to processing an output report in the Buttons and Lights example. One complication, however, is that we do not know where in the frame that our data will be so we cannot process it in real time. We need to buffer the data from the current frame while processing the data from the previous frame. We will synchronize with the SOF packet and generate timer interrupts every 64 usec to output the other 15 data bytes.

If a packet is missed then we will repeat the last sample 16 more times – we must always output to the DAC every 64 usec or we will hear pops and cracks. Early versions of the Windows 98 operating system did drop packets since the USB audio driver was running at a low priority level. This has been fixed in Windows ME and Windows 2000 by running the audio subsystem at the kernel level.

So the only firmware we really needed to change was the SOF interrupt handler, and the TIMER interrupt handler. These changes are shown in Figure 12-7 and the full program source is available on the CDROM.

Figure 12-7. New interrupt handlers for sound.

The prototype hardware used in this example is shown in Figure 12-8. Also in Figure 12-8 is a screen shot of the Windows Control Panel applet showing that this device enumerated as a standard audio device that was integrated into the Windows sound subsystem.

Figure 12-8 The example integrated into the Windows sound subsystem.

Upgrading to 16-bit Stereo

Our initial 8-bit mono design is a great way to add speech or simple sounds output to any EZ-USB design. The hardware cost was small and the software overhead was manageable. The second example is a small move to 16-bit stereo. We shall see that the software impact is small but the hardware is more complex.

We need only change the setup values in two descriptors, as shown in Figure 12-9, to identify ourselves as a 16-bit stereo device to the PC host. The PC host will now send us four times as much data per frame that we need to meter out to two 16-bit DACs.

Figure 12-9. Descriptor changes for 16-bit stereo.

Building stereo USB speakers is very easy because almost all of the work is already done for us in the operating system and via the USB Specification. The bytes are already ordered exactly as they are needed to be sent to the DAC. One hardware wrinkle in the design is the availability of DACs for stereo audio applications. All of the available components are manufactured with a serial interface designed to operate gluelessly with a DSP, and they include ADC input channels (that is, they are coder/decoders—codecs). The interface is too fast for a microcontroller to "bit-bang." Parallel interface stereo audio DACs are available, but these components also include analog inputs and other mixing circuitry. So our choice is a serial codec (such as the AD1849) and a parallel-to-serial converter with a PAL controller, or a parallel codec (such as the AD1845). Figure 12-10 shows a potential hardware implementation using an Anchor Chips device and a serial codec.

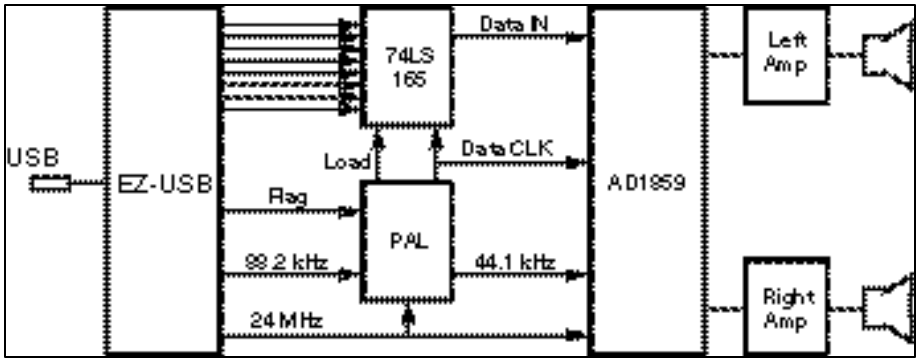


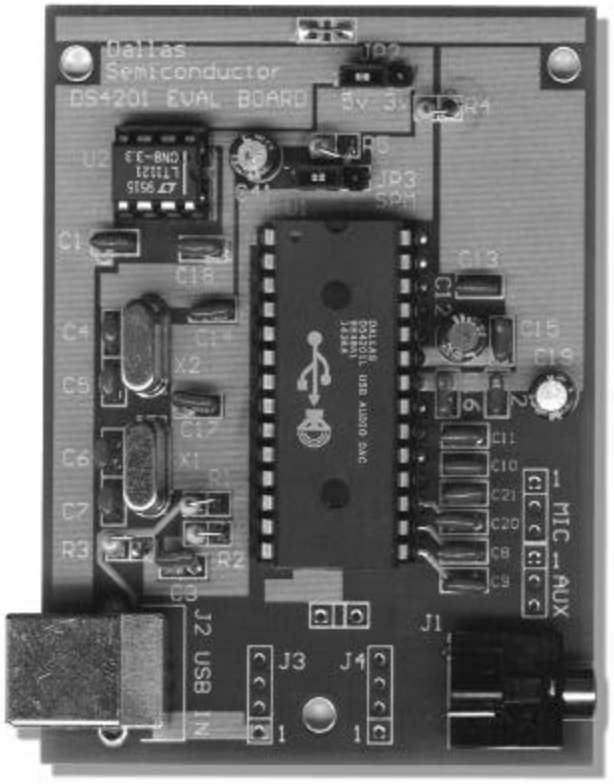
Figure 1-1. Discrete implementation of USB speakers

The hardware is getting a little cumbersome. I did cheat a little in this example – I kept the sample rate at 16KHz since this is easy for the EZ-USB microcontroller to decode. Real 16-bit stereo systems operate at 44.1KHz and this creates a great deal of problems for the EZ-USB solution (or any other general purpose microcontroller for that matter).

A rate of 44.1KHz results in the PC host sending a varying number of bytes per frame (since the samples don't neatly fit into 1 msec). It also increases the amount of data that the microcontroller must handle by almost 300%. More difficult to handle however, is the long term stability of the output signal. One presumes that 44.1KHz 16-bit stereo is being used to listen to long music segments – the synchronization of the microcontroller output rate with the true sample rate must now be allowed for. The USB specification includes several mechanisms that may be used by an I/O device to compensate for clock drifts and other disrupting effects. The I/O device must provide a feedback endpoint to the PC host audio driver to adjust its data rate. This is becoming too complex for a general purpose microcontroller solution and it is time to turn our attention to hardware designed to support all of the features of high quality audio.

DEDICATED SPEAKER SOLUTION

Dallas Semiconductor has integrated the USB transceivers, the SIE, the control, and the DACs onto a single component (Figure 12-11). The internal design is not available to change as it is in the previous example—the traditional tradeoff between flexibility and simplicity. The Dallas DS4201 also includes analog inputs that may be mixed with the digital audio from the PC (Figure 12-12), but these features are not used in this first example.



Courtesy of Dallas Semiconductor Corp.

Figure 1-2. Single-chip implementation of USB speakers

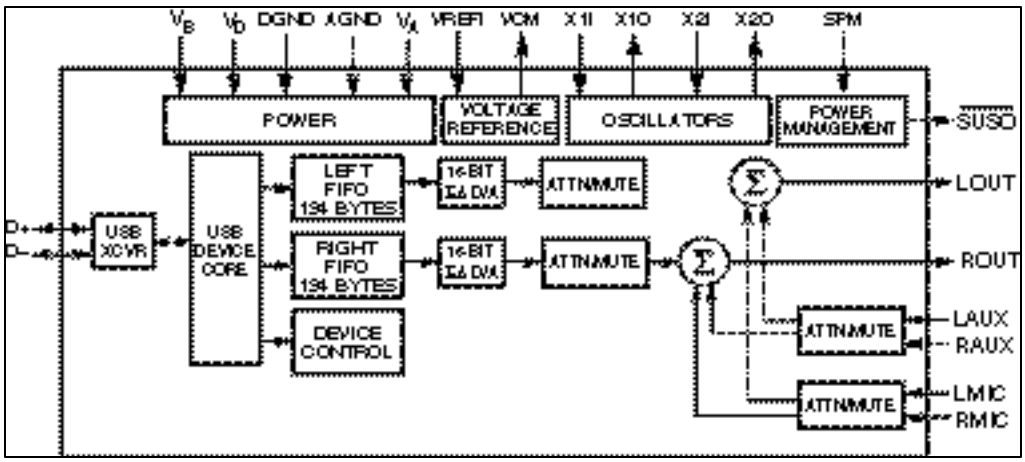


Figure 12-12. Block diagram of the DS4201

After uniquely identifying each of the terminals and units, a descriptor is written for each item and declared alongside the other USB descriptors. The full set of descriptors for the DS4201 is included on the CD-ROM under Chapter 12/Dallas Semiconductor/DS4201 Descriptors.pdf.

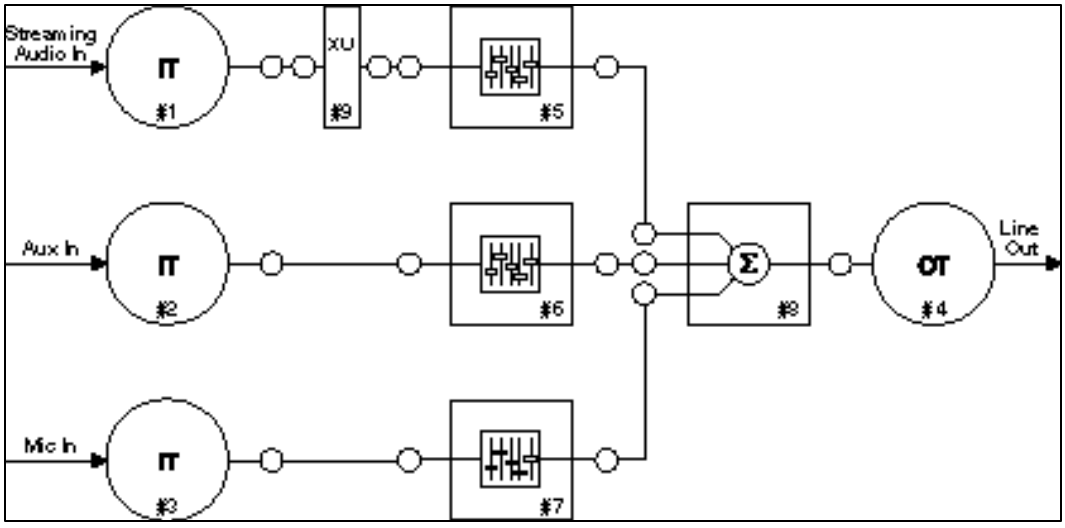


Figure 12-13. Software model of the DS4201

Audio Input via a Microphone

It is straightforward to convert our EZ-USB speaker solution into an audio input device using a microphone. The hardware, shown in Figure 12-14, uses a pre-amplifier and an analog-to-digital converter. The firmware samples the input every 64msec and creates a 16 byte buffer each frame. The EZ-USB component supports double buffering on endpoints so that data can be collected in the current frame while the PC host is reading data from the previous frame. The descriptors must be altered to define INput data rather than OUTput data. The full source code is provided on the companion CDROM.

Figure 12-14. Circuitry used for microphone input

The same issues arise when upgrading to 16 bit stereo as with the speakers example. A sample rate of 16KHz is easy to support but if you need 44.1KHz then it is time to look at special purpose hardware.

An additional facet of inputting audio is the determination of the correct input signal. A typical background environment will have noises that the microphone will pick up just as it picks up the speech signal. This has been solved in the past by using directional microphones that are attached to the talker typically on a headset or a throat microphone.

USB enables a new category of audio input devices that can determine the background noise and adapt to the position of the talker. Different talkers can be identified and even tracked as they move around a room. Figure 12-15 shows a microphone array from Andrea Electronics that may be fixed to a PC host or a video conferencing system so that the talkers do not need to be attached to microphones.

Figure 12-15. Digital Super Directional Array technology from Andrea Electronics

The key to the microphone array's performance is the DSDA microphone's ability to focus on and enhance desired acoustic signals while electronically eliminating ambient noise sources. The inputs from the array of microphones are fed into the PC host which analyses the information from each of them and creates an adaptive focus beam which it "directs" towards the talker. The array constantly samples the acoustic environment, adaptively identifying interfering noises that are extraneous to the signal path. In a video conferencing application it can also be used to direct and focus the camera to the current talker.

CD QUALITY SOUND INPUT AND OUTPUT

In this section, we'll describe a bi-directional USB audio controller from Texas Instruments, the TUSB3200 USB streaming controller (STC). This was specifically designed for applications requiring isochronous data streaming. Targeted applications include digital speakers, headsets, microphones, or any other application where isochronous data over USB is required.

TUSB3200 Architecture

Although STC is mainly designed for audio streaming applications, there is a fair amount of flexibility in the architecture allowing it to be used in almost any USB application requiring data streaming. The figure 12-16 shows the block diagram of a TUSB3200.

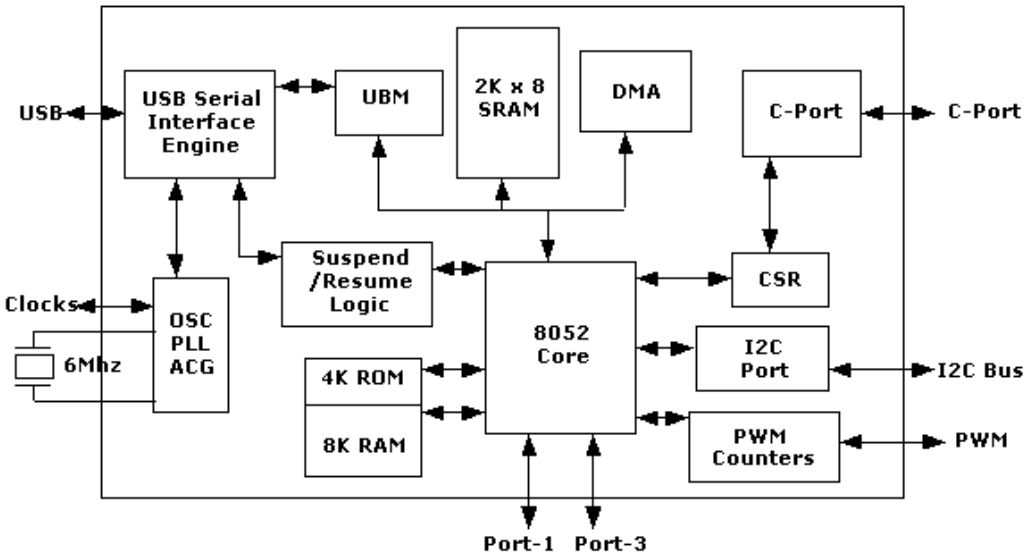


Figure 12-16. TUSB3200 Block Diagram

The TUSB3200 has a standard 8052 microcontroller unit (MCU) core with on-chip memory. The MCU memory includes 4K bytes of memory ROM that contains a boot loader program which downloads the application firmware at initialization from a nonvolatile memory on the PCB to an 8K RAM on-chip. It supports 12 Mb/s (full speed) data transfers and has seven IN-endpoints and seven OUT-endpoints that are fully configurable by the MCU application code. The configuration of ISO endpoints allows simultaneous record and playback. The CODEC interface of the device supports audio CODEC (AC) ’97 and several inter-IC (I2S) modes. The I2S bus was developed to efficiently move digital sound data between components. The DMA controller with four channels is provided for streaming the USB isochronous data packets to/from the CODEC port interface. An on-chip phase-locked loop (PLL) and adaptive clock generator (ACG) are used for the USB synchronization modes.

TUSB3200 Evaluation Module

The TUSB3200 Evaluation Module (EVM) comes with everything a developer needs to start designing USB audio peripherals. The EVM is designed to allow quick development and evaluation of the TUSB3200 device for audio streaming applications. There are no custom drivers required for the EVM. The evaluation module is fully functional and can be plugged into a USB port for audio streaming right away. Windows USB Audio drivers is all that is needed for full functionality. The figure 12-17 shows a photograph of the evaluation module, and figure 12-18 shows the block diagram of the module.

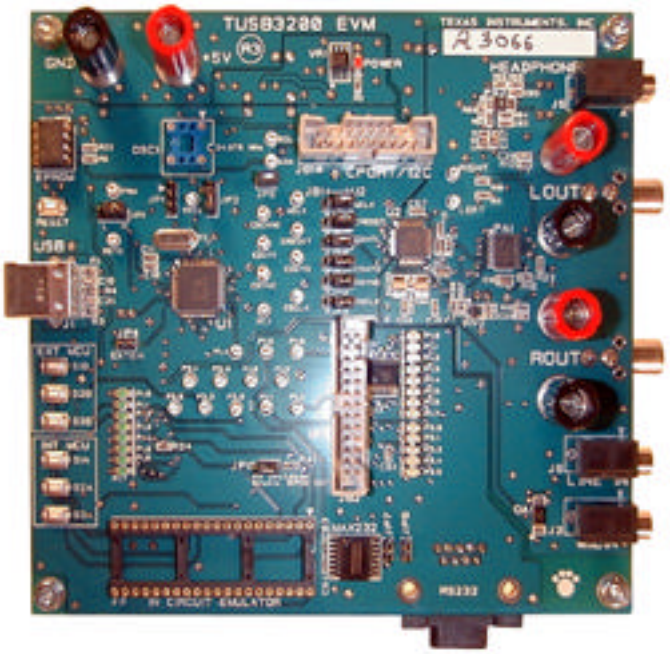


Figure 12-17. TUSB3200 Evaluation Module

The data moving is handled by the dedicated DMA channels programmed by the application code. The internal MCS52 microcontroller implements the higher-level USB protocols. The controller on the PCB accepts USB data streams and generates output at the CODEC Port (CPORT), I2C port and or GPIO port.

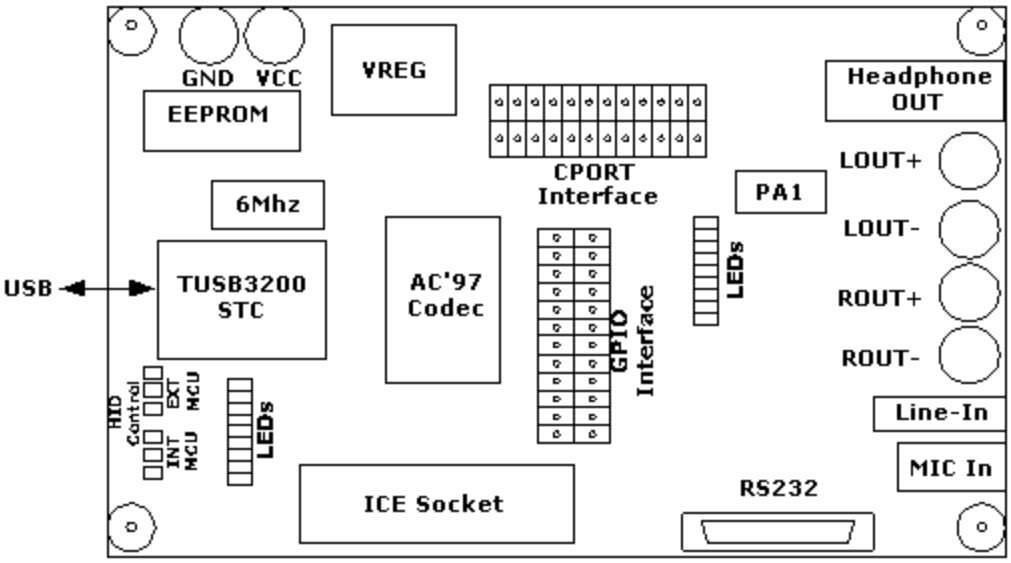


Figure 12-18. TUSB3200 EVM Block Diagram

Development Tools

The EVM has an 8052 ICE socket for firmware development and debug. Any third party 8052 emulator can be used. Although USB bus analyzer is optional, it is highly recommended to reduce development effort. If you prefer to program in C, you will need an 8052 C-Compiler. For development environment, an In-Circuit emulator and the USB interface are the only connections required. All other connections are used for increased functionality to allow further development and evaluation.

Getting Started with TUSB3200 Programming

The programming of STC can be done either in C or assembly. Although writing programs in C would require less effort, the efficiency achieved when doing so is less than what you can achieve by writing assembly programs. Efficiency here means having as few instructions or as few clock cycles as possible. The programmer should also obtain a copy of USB Specification and USB Class Definition for Audio Devices, Data formats, Human Interface Devices and Terminal Types. The baseline source code for STC can be obtained from Texas

Instruments Inc. by sending an email to digitalaudio@ti.com (under free licensing agreement). Here we will only show the steps required to program TUSB3200 at a higher level.

Sample Pseudo-Code for STC Initialization

All programs start by going through a reset initialization code. Upon power up, the system always goes to the reset location in memory and the internal bootloader downloads the firmware stored on EEPROM.

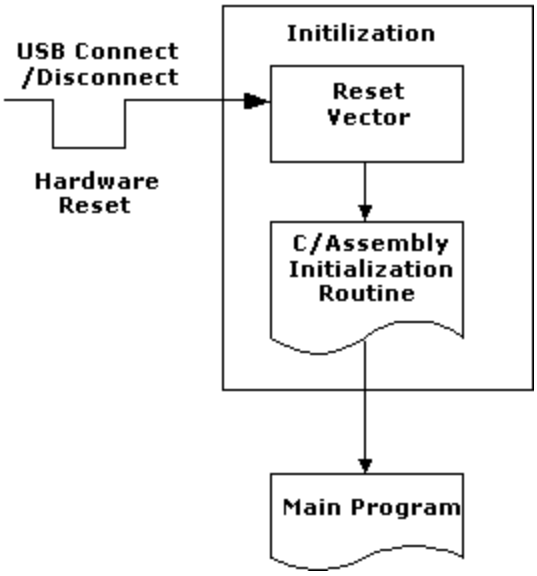


Figure 12-19. Device Initialization

Once the application code download is complete, on-chip 8052 executes the code to set up the system for proper operation. The application firmware for STC consists of three main parts: “USBEngine”, “ACGInit” and “DeviceInit”. USB Engine is where the USB protocol is implemented. This module is responsible for sending and receiving USB data packets. ACG initialization routine sets up the on-chip adaptive clock generator (ACG). The programming of ACG depends on the desired sampling frequency. The “DeviceInit” routine initializes the rest of the device registers. The TUSB3200EVM is a USB composite device including a speaker, a microphone, a line-in, and a mixer. The device supports

16 bit stereo PCM Type-1 stream out from host for playback and stream in to host for recording. The speaker unit provides feature controls for the left/right channel, master mute, bass, and treble. The microphone functionality provides feature controls for microphone sidetone and recording. The volume controls consist of master volume and mute. The Line-In functionality provides feature control for left/right channel and the master mute. The mixer unit is used to sum up the audio stream from host, the microphone feedback, and the line-in. The sum is input to the feature control unit of the speaker. The following figure shows the USB topology for the TUSB3200EVM.

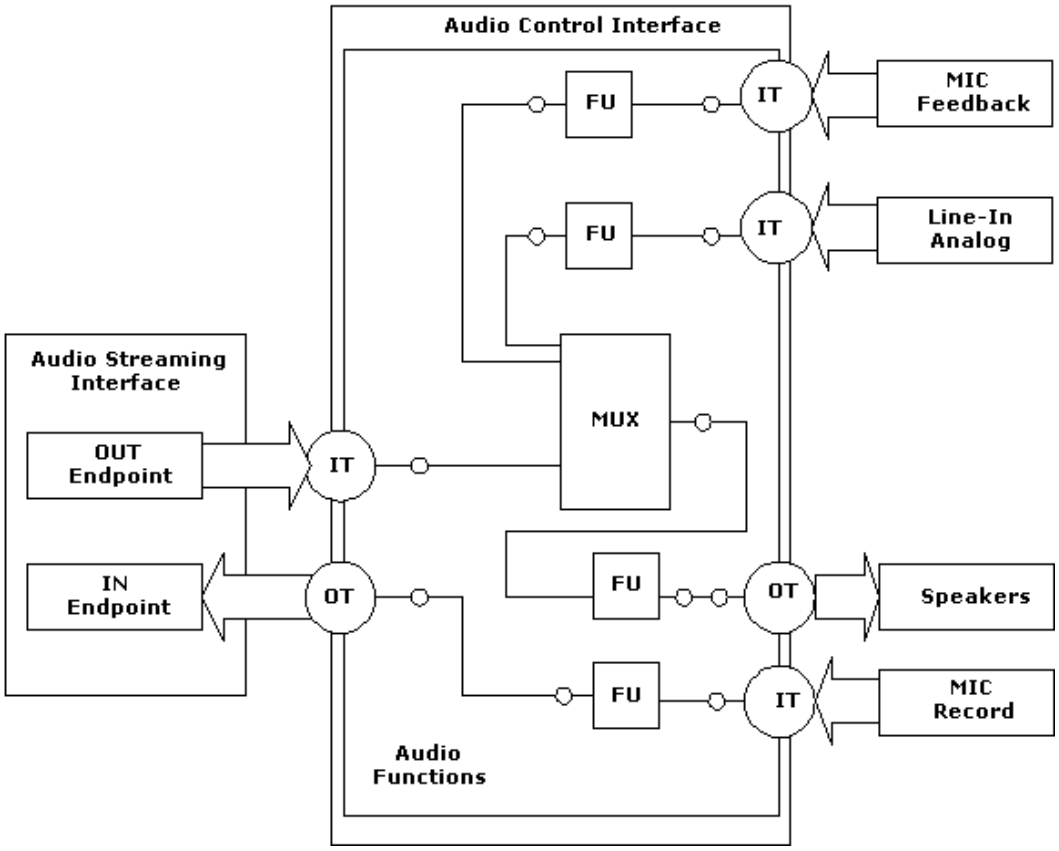


Figure 12-20. TUSB3200EVM USB Audio Topology

The device descriptors for the TUSB3200EVM is based upon the topology given above. The HID descriptor is not part of the USB Audio specs; however, its descriptors are added at the end of the USB Audio descriptor hierarchy for convenience. Figure 12-21 shows the descriptor hierarchy for the TUSB3200EVM:

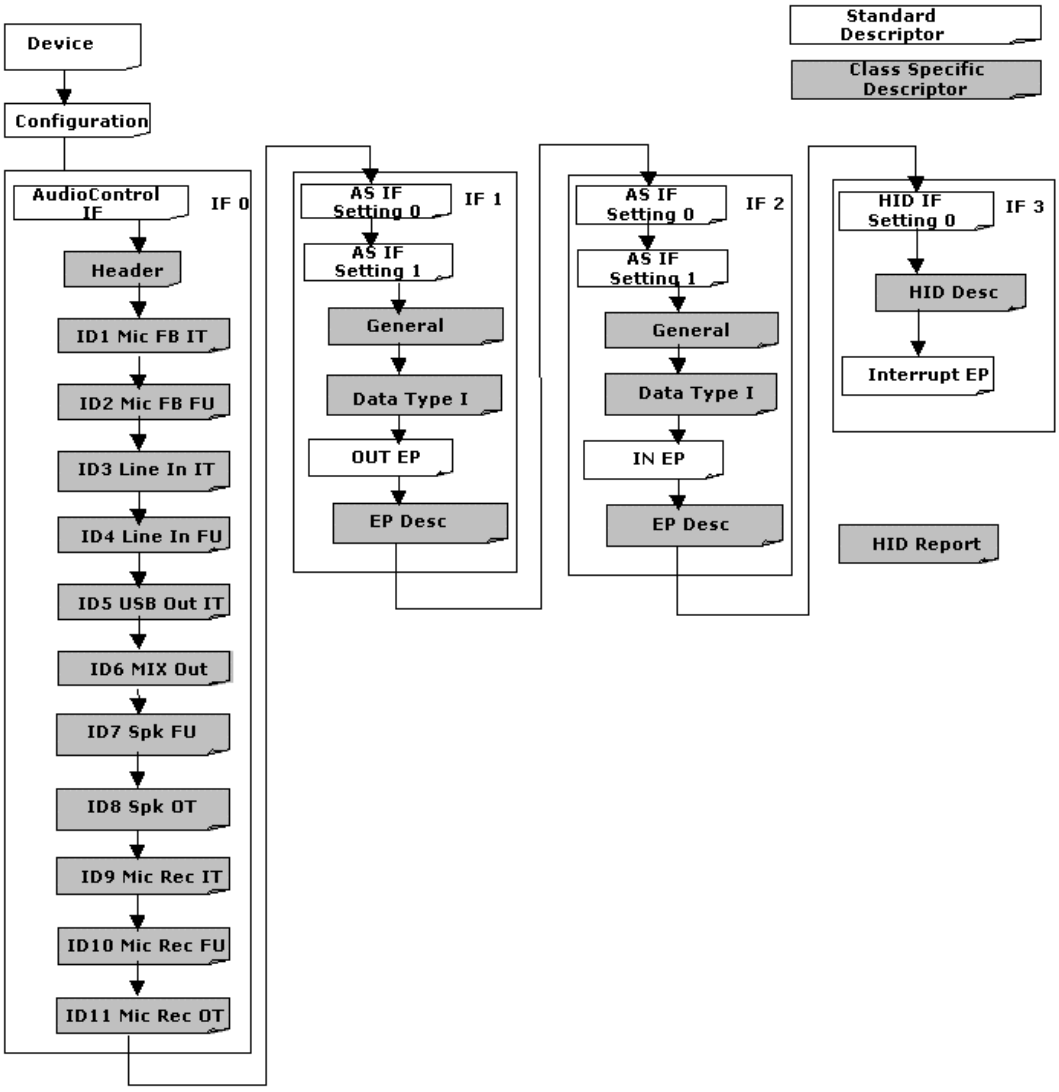


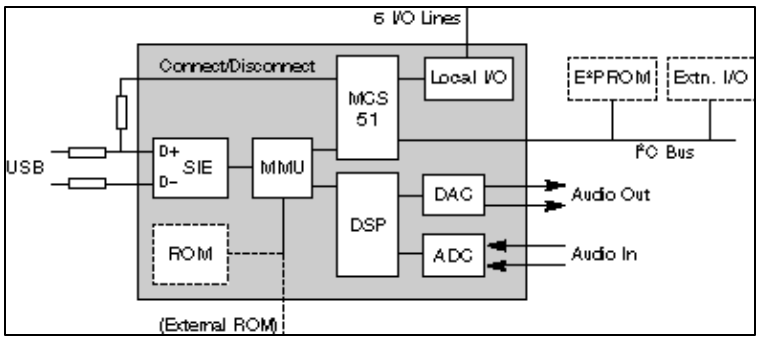
Figure 12-21. TUSB3200EVM Descriptor Hierarchy

AN INTERNET TELEPHONE

The fundamental design problem of connecting a microphone and speaker together into a single device to form a telephone is that it has the same circuit diagram as an oscillator; sound from the speaker can be coupled to the microphone to generate a feedback path. A pair of connected telephones (Figure 12-22) produces several undesirable signal paths which must be reduced to create a usable product.

Figure 12-22. Undesirable signal paths with connected telephones

Two echoes (near-end and far-end) and the detection of both talkers active at the same time, can be eliminated by some digital signal processing. The caller perceives an annoying acoustic echo due to near-end coupling and hears their own voice with a delay depending upon the Internet and the global system delay due to far-end coupling. The length of the far-end echo path varies with each connection so any echo cancelling algorithm must be adaptive. The DSP algorithms can execute on the PC host or in the I/O device. Figure 12-23 shows a specially designed component, from ZMM, to solve this real-world noise cancellation problem. The DA040 includes an MCS51 microcontroller, USB and audio interfaces AND an embedded 50-MIP digital signal processor.



Courtesy of ZMM.

Figure 12-23. USB-audio component from ZMM includes a DSP

ZMM have a reference design that builds a digital handset using the base components of a standard telephone plus their DA040. The full schematics are included on the CDROM and a photograph of their "Chat Phone*" is shown in Figure 12-24. The Chat Phone* plugs into any available USB port and will be recognized as a set of standard audio class devices by the Windows operating system. If the PC host is connected to the Internet via a modem and phone line there is no need to break the connection to make an Internet phone call- the PC host will route the digital audio from the USB phone via the same Internet connection. Dialing with Chat Phone* can be achieved by voice commands, screen dialing or from any Personal Information Manager (PIM).

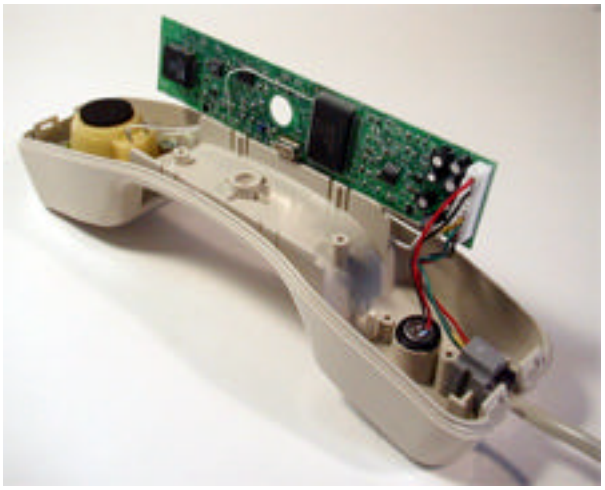


Figure 12-24. Digital handset reference design

Desktop Business Audio

While waiting for the planet to convert to Internet telephone calling there is a requirement to connect to the current telephone lines.

The uniform treatment of digital sound within the USB architecture makes it easy to build an I/O device that is a combination of buttons, lights, and sounds. The telephone has been the number one tool for increasing business productivity for a long time. USB makes it easy to integrate it with the number two device—the PC platform. Together they can create a tool that should exceed our productivity expectations.

To get us in the right frame of mind, please don't think of this as adding a telephone to a PC computer. This will limit our imagination to obvious conclusions such as "well, the PC could do the dialing, I suppose." It is much better to think that you are adding a powerful PC to a telephone—what more could the telephone do if it had a 800-MHz Pentium III processor, 128-MB memory, and a 42-GB disk inside it?

Possibilities for an improved telephone now start to open up. Today's PC can easily do speech recognition, so saying "computer, dial Bob" is deliverable today. .

Figure 12-25 shows the design problem of a USB-based telephone divided into elements that we already have solutions for. The Data Access Arrangement (DAA) will vary from phone system to phone system, and there are many restrictions relating to phone-line connected equipment. These regulations vary by country; local documents can be obtained that define the scope and testing of a particular telephone authority interface.

ZMM have a reference design that supports all of the features that you think you may need (and several more that you didn't think of) in a Desktop Business Audio solution. The full schematics are included on the CDROM and a photograph of their design is shown in Figure 12-26. The central core of the design is a DA050 component which is a DA040 with different microcontroller and DSP firmware. In fact, ZMM have a whole range of products based on this core ranging from toys to high end audio equipment – they will also help you customize their part for your design (if you buy enough of them).

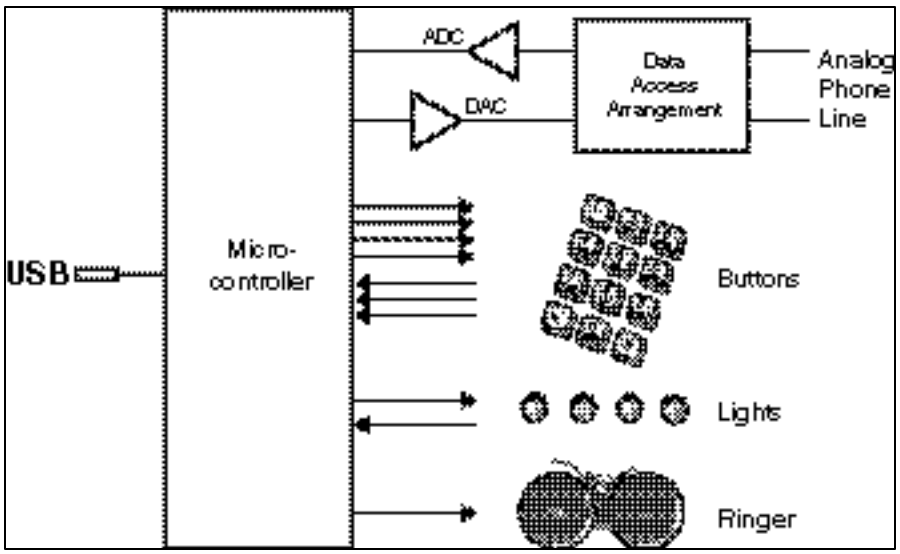


Figure 12-25. Major elements of USB-based telephone

Figure 12-26. Desktop Business Audio reference design

The reference design interconnects the telephone line, microphone, headset (or speakers) using USB. It also includes the DAA for the phone line so that a modem and even fax machine can be supported with just software. ZMM have targeted their products to have worldwide appeal and they have been certified as passing many telephone standards as shown in Figure 12-27. The reference design supports call recording and logging on the PC host and can generate and detect DTMF signals. It supports all of Microsoft's interfaces to allow voice mails to be integrated into MS OUTLOOK.

Figure 12-27. DAA's across the world are supported

MIDI OVERVIEW

The growth of MIDI has been hampered by lack of standardization. Fortunately today, thanks mainly to the publication of the General MIDI System specification and its inclusion in the Windows operating system, MIDI is seeing widespread use in all aspects of the PC platform.

MIDI is fundamentally synthesized sound. Figure 12-28 shows a waveform of a typical synthesized sound waveform. It has a start, a middle, and an end. The middle consists of four sections—the initial sound or attack, an early loss of initial amplitude or decay, a continuance of sound or sustain, and finally a release. The richness of sounds is described by this envelope; a simple on-off sine wave is not representative of real-world sounds.

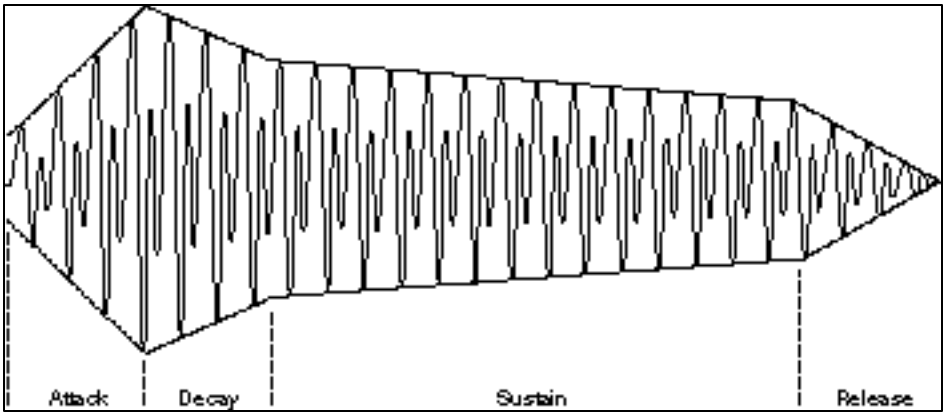


Figure 12-28. A typical synthesized sound waveform

The waveform in Figure 12-28 can be represented in two MIDI messages, each of which is only three bytes' worth of data! The MIDI protocol is a very efficient method of representing musical performance information. The de facto standard for MIDI was defined by the MPU 401 from the Roland Corporation (now Edirol). This was an I/O port interface. A synthesizer is required to translate MIDI messages into sounds. Although the original synthesizer was implemented in hardware on a Sound Blaster board, this is not required. The synthesizer can be implemented in hardware or software, inside or outside the PC host. The Windows operating system includes a hardware-independent device driver that allows different hardware implementations with no change to the application software.

Looking deeper inside MIDI, we discover that there are actually four components: the MIDI communications protocol, the MIDI stored file format, the MIDI synthesizer, and the MIDI hardware interface.

MIDI Protocol

MIDI information is transmitted in MIDI messages, which can be thought of as instructions that tell the synthesizer how to play a piece of music. The synthesizer receiving the MIDI data generates the actual sounds. MIDI messages are transmitted as a serial bit stream at 31.25 Kbps with 10 bits transmitted per byte (a start bit, 8 data bits, and one stop bit). Why 31.25 Kbps? The original hardware divided the 8-MHz ISA bus clock signal by 256! The protocol divides the single physical MIDI channel into 16 logical channels that are controlled independently. A MIDI message is made up of a status byte that is generally followed by one or two data bytes. At the highest level, MIDI messages are classified as being either Channel Messages, which are targeted for a logical channel, or System Messages, which control the entire performance.

The MIDI protocol defines the activation of a particular note and the release of the same note as two separate events. For example, when a key is pressed on a MIDI keyboard, it transmits a **Note On** message on one preprogramed MIDI channel. The Note On status byte is followed by two data bytes, which specify key number (indicating which key was pressed) and velocity (how hard the key was pressed). When the key is released, the keyboard will send a **Note Off** message. The Note Off message also includes data bytes for the key number and the velocity.

At the beginning of a MIDI sequence, system messages are sent on each channel used in the performance to set up the appropriate instrument sound for each part. The General MIDI Specification includes the definition of a General MIDI Sound Set (a patch map), a General MIDI Percussion map (mapping of percussion sounds to note numbers), and a set of General MIDI Performance capabilities (number of voices, types of MIDI messages recognized, etc.). A MIDI sequence that complies with the General MIDI Specification will play correctly on any General MIDI synthesizer or sound module.

MIDI File Format

MIDI data files are very small when compared with PCM data files (that is, .WAV files). For instance, PCM files require about 10 MB for each minute of CD-quality audio while a typical MIDI file might consume less than 10 KB for each minute. This reduction is possible because the MIDI file contains only the

instructions required to re-create the sounds and not the sounds themselves. A synthesizer will generate the actual sounds from these instructions.

The International MIDI Association publishes a Standard MIDI Files Specification that defines how time-stamped MIDI data should be stored. The interested reader can purchase this specification from www.midi.org (they would not provide a copy for inclusion on the CD-ROM). This standard allows different applications such as sequencers, scoring packages, and multimedia presentation software to share MIDI data files.

MIDI Synthesizer

Three methods are currently used to synthesize sounds: frequency modulation, wavetable, and physical modeling.

Frequency Modulation, or FM, synthesis involves combining fundamental sine waves of different frequencies and amplitudes to create a complex waveform. FM synthesis is a good technique for beeps, bangs, explosions, and space-type noises but is not suitable for creating the richness of a musical instrument.

Wavetable synthesis starts with actual music instruments; these are recorded, and short digitized sections are stored. When an application program requests a particular sound, a sample is retrieved, processed, played back, and typically looped. Wavetable sound quality is generally very good, provided the original recordings are of high quality.

Physical modeling is a combination of physics and art; a mathematical model that takes into account all of the forces on, say, a violin string, is used to predict the sound that plucking or bowing the string would make. The result is an incredibly lifelike reproduction, though the initial coding of each model is time-consuming.

The output of a synthesizer is sound; this could be analog sound in the case of a hardware implementation of a synthesizer, or it could be digital sound, high-resolution PCM format, for a software implementation. Hardware implementations have been popular in the past, but the increased performance of today's PC platform processor allows the processor to do a superior job for lower system costs.

MIDI Hardware Interface

The first implementation of a MIDI interface on a PC platform used the game port. I don't know why—that would appear to be the worst choice, because this port was designed to interface to analog joysticks. The only valuable attributes I could discover for this port were that it was available and FREE—it was included on all early PC platforms, and few people used it. A lot of software, and processor bandwidth, is required to send and receive serial messages on this parallel port. Compared with the MIDI protocol, however, the processor has lots of time to do this, and if the PC platform isn't doing anything else, then there's no problem. But as people wanted the processor to do more things, the "resource-wasting" software had to be replaced by a serial controller, typically on the sound card.

The modern PC does not have a joystick port—it was removed in the *PC 99 System Design Guide* because its burden on system performance was becoming excessive.

The modularity and layered implementation of the Windows operating system allows a MIDI application's system calls to the MIDI hardware to be intercepted and redirected. The Roland Corporation supplies a device driver that redirects MIDI requests to USB. A USB subsystem, called the S-MPU64, receives these requests and drives the real MIDI ports. The S-MPU64 includes 4 MIDI IN connections and 4 MIDI OUT connections (Figure**Error! Reference source not found.**). This entry-level system is powered by the USB port and provides the connectivity required for a range of MIDI equipment.

Courtesy of Edirol Corp.

Figure 12-29. USB-to-MIDI interface from Edirol (Roland)

Roland has recently introduced a "big brother" to the S-MPU64 that includes multiple audio connections. Roland included audio line input and output, and local guitar and microphone inputs, on their UA-100 product (Figure 12-30) to create a complete digital audio subsystem.

Courtesy of Edirol Corp.

Figure 12-30. USB-based audio subsystem from Edirol (Roland)

The Roland UA-100 enumerates as an Audio/HID device with four separate functions. A block diagram (12-31) highlights the major elements connected to a single USB cable.

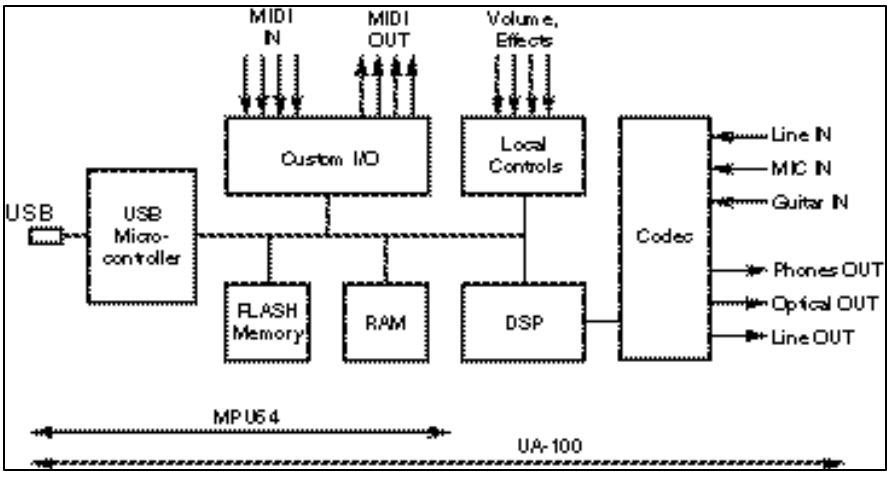


Figure 12-31 Block diagram of Edirol's audio subsystem

USB-ENHANCED MIDI

MIDI is a mature standard with many existing products and applications. It is a standard that defines not only the data protocol for exchange of musical control information but also the hardware connection, used to physically exchange the data. Therefore, transferring MIDI data over another hardware connection like USB is not really conforming to the MIDI specification and is therefore called USB-MIDI. MIDI exchange over USB is designed to be an elegant method to enable a wide range of MIDI system configurations, from the simplest to the most complex. Furthermore, USB-MIDI expands on MIDI, allowing applications not possible with non-USB MIDI configurations. USB is well suited for connecting MIDI Interfaces and MIDI instruments to computers. USB-MIDI builds on the strengths of MIDI by adding higher speed of transfer and increased MIDI channels through its multiple "virtual" cable support.

USB transfers MIDI data at rate hundreds of times faster than the original MIDI 1.0 hardware specification. In addition, USB-MIDI takes advantage of the USB's bandwidth and flexible data handling to enable the transfer of many "virtual" cables worth of MIDI data. Multiport MIDI interfaces have become more commonplace today and they need a connection to the computer that can handle multiple MIDI connections on one cable. USB is very well suited to this task.

Synthesizers and other MIDI instruments have increased in abilities. The bandwidth of a traditional MIDI connection can be more easily consumed when trying to serve the high polyphony and increasing number of MIDI message types commonly used. Typical synthesizers now also use the 16 MIDI channels available on a MIDI bus in one instrument alone, requiring multiple MIDI busses in a typical setup with more than one MIDI instrument. Additionally, timing accuracy is essential in music. USB can easily handle heavy loads of MIDI data while preserving the timing integrity of the data. Hundreds of MIDI note messages can be sent all at the same time. In addition, by handling multiple "virtual" cables the USB offers a solution to go beyond MIDI's 16-channel limit.

MIDI data is transferred over USB using 32-bit USB-MIDI Event Packets. These packets provide an efficient method to transfer multiple MIDI streams with fixed length messages. The 32-bit USB-MIDI Event Packet allows multiple "virtual MIDI cables" routed over the same USB endpoint. This approach minimizes the number of required endpoints. It also makes parsing MIDI events easier by packetizing the separate bytes of a MIDI event into one parsed USB-MIDI event.

Figure 12-32 shows the layout of an Event Packet. The first byte in each 32-bit USB-MIDI Event Packet is a Packet Header contains a Cable Number (4 bits) followed by a Code Index Number (4 bits). The remaining three bytes contain the

actual MIDI event. Most typical parsed MIDI events are two or three bytes in length. Unused bytes must be padded with zeros (in the case of a one- or two-byte MIDI event) to preserve the 32-bit fixed length of the USB-MIDI Event Packet.

Figure 12-32 32-bit USB-MIDI Event Packet

The USB-MIDI function exposes a single MIDIStreaming interface that is used by Host software to interact and control the entire MIDI functionality of the function. Since all control messages are performed in-band (buried into the MIDI data stream) there is no need for a separate control and status interface. The MIDIStreaming interface contains one or more MIDI endpoints, which all use bulk transfers to exchange MIDI data with the Host. In addition, one or more Transfer bulk endpoints can be present to provide a high bandwidth path to some of the Elements inside the USB-MIDI function.

CHAPTER SUMMARY

The features built into USB enable it to support sound as easily as it supports other data types. Sound uses the isochronous capability of USB to implement real-time data delivery. The operating system already includes all of the software drivers required to implement a high-quality, digital audio system—this means that we have no PC host software to write (always a good sign). Simple audio designs can be implemented on a general purpose USB microcontroller such as the EZ-USB; the task consists of implementing the descriptors that define the device and its features, and writing a service subroutine for the isochronous interrupts. For higher quality sound input and output, one of the prebuilt audio devices should be used. The modular, building-block structure of USB allows sound to be easily added to most USB I/O devices.